

Soluzione

Esercizio 2:

Scrivere un nuovo tipo di dato coppia di interi.

Inizializzare tre istanze di coppie con i primi tre numeri naturali e i loro doppi.

Soluzione:

```
#include <stdio.h>

typedef struct coppia {
    int x;
    int y;
} Coppia;

main(){
    Coppia a, b ,c;

    a.x=1;
    a.y=2;

    b.x=2;
    b.y=4;

    c.x=3;
    c.y=6;

}
```

Esercizio 3:

Allocare dinamicamente tre istanze del tipo di dato coppia appena definito, inizializzarle con i valori dei primi tre numeri primi e dei loro quadrati e quindi dellaocarle.

Soluzione:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct coppia {
    int x;
    int y;
} Coppia;

main(){
    Coppia *a, *b ,*c;

    a = malloc(sizeof(Coppia));
    b = malloc(sizeof(Coppia));
```

```

c = malloc(sizeof(Coppia));

/* a,b e c sono puntatori, quindi prima di accedere ai campi della struct
i puntatori vanno dereferenziati.
Notate le parentesi, necessarie perche' l'operatore di dereferenziazione (*)
ha priorita' minore dell'operatore accesso alla struttura (.) */
(*a).x=1;
(*a).y=1;

/* oppure possiamo usare l'operatore freccia -> che esegue le due cose insieme
e senza la necessita' di usare parentesi. */
b->x=2;
b->y=4;

c->x=3;
c->y=9;

free(a);
free(b);
free(c);
}

```

Esercizio 4:

Scrivere una funzione che riceva un array di coppie di interi e le inizializzi con i primi n numeri dispari (n e' la lunghezza del vettore) e i loro quadrati.

Soluzione:

```

#include <stdio.h>
#include <stdlib.h>

typedef struct coppia {
    int x;
    int y;
} Coppia;

/*
Dopo la typedef, Coppia e' un tipo di dato completamente legittimo e puo' essere
usato nello stesso modo con cui si usano i tipi standard.
*/
void initCoppie(Coppia arr[], int dim) {
    int i;
    for(i=0; i<dim; i++) {
        /* Ovviamente arr[i] ha tipo Coppia quindi va trattato come tale */
        arr[i].x = i+1;
        arr[i].y = (i+1)*(i+1);
    }
}

main(){
    int i;
    Coppia vett[10];

    initCoppie(vett,10);
}

```

```

    for(i=0; i<10; i++)
        printf("(%d, %d)\n", vett[i].x, vett[i].y);
}

```

Esercizio 6:

Scrivere una funzione ricorsiva che riceva un array di interi e lo ordini, usando l'algoritmo di Merge Sort.

Soluzione:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define LUNG 100

void mergesort(int vet[], int dim);

main() {
    int vett[LUNG], i, n;
    srand(time(NULL));

    printf("Inserire la lunghezza dell'array (max %d): ", LUNG);
    i=scanf("%d", &n);
    if(i==0 || n>LUNG) return;

    for(i=0; i<n; i++) {
        vett[i]=rand()%100;
        printf("%d ", vett[i]);
    }
    printf("\n");

    mergesort(vett, n);

    for(i=0; i<n; i++)
        printf("%d ", vett[i]);
    printf("\n");
}

/*
Funzione per eseguire lo shift verso destra degli elementi di un array.
Ritorna l'elemento che "sfora" ossia l'ultimo elemento dell'array originale
Il primo elemento rimane immutato (e quindi e' presente in due copie dopo lo
shift).
*/
int shift_right(int vet[], int dim) {
    int i, temp=vet[dim-1];
    for(i=dim-1; i>0; i--)
        vet[i]=vet[i-1];
    return temp;
}

```

```

/*
La funzione merge esegue la seconda parte dell'algoritmo di mergesort, la
riunificazione dei due sotto-array ordinati in un'unico array ordinato.
Notare che:
- poiche' i due sotto array sono ordinati, l'elemento piu' piccolo in assoluto
e' uno dei due elementi in testa ai sotto-array
- poiche' dobbiamo fare spazio per il nuovo array che stiamo compilando,
quando l'elemento piu' piccolo e' il primo del secondo array, dobbiamo shiftare
tutto il primo array verso destra (con la funzione precedente)
- quando invece l'elemento piu' piccolo e' in testa al primo array, allora e'
gia' in posizione e basta ridurre le dimensioni del primo array
*/
void merge(int vet1[], int dim1, int vet2[], int dim2) {
    int temp;

    if(dim1==0 || dim2==0) return;

    if(vet2[0]<vet1[0]){
        /* l'elemento piu' piccolo e' in testa al secondo segmento */

        /* quindi lo mettiamo in testa a tutto e shiftiamo il primo segmento a destra.
        Questo funziona perche' i due segmenti sono contigui.
        */
        temp=shift_right(vet1, dim1); /* in temp ci viene l'elemento che prima era in
fondo a vet1 */
        vet1[0]=vet2[0];
        vet2[0]=temp;

        /* si riduce il secondo segmento e il primo resta uguale ma shiftato a destra
        */
        merge(vet1+1, dim1, vet2+1, dim2-1);
    }
    else {
        /* l'elemento piu' piccolo e' gia' in testa al primo segmento
        quindi si riduce il primo segmento e il secondo resta uguale */
        merge(vet1+1, dim1-1, vet2, dim2);
    }
}

/*
Banale funzione di divisione: ogni volta si suddivide l'array in due pezzi
uguali si ricorre su questi.
Quando le due chiamate ricorsive ritornano, e' garantito che i due sotto-array
siano ordinati e quindi basta completare il tutto con una chiamata a merge
che sistema i due sotto array.
*/
void mergesort(int vet[], int dim) {
    int i;

    if(dim<2) return;

    i=dim/2;

    mergesort(vet, i);
    mergesort(vet+i, i+(dim%2));

    merge(vet, i, vet+i, i+(dim%2));
}

```

```
}
```

Esercizio 9:

Scrivere un programma che crei dinamicamente una lista di 3 interi e li inizializzi ai primi tre naturali.

Il programma deve deallocare correttamente la lista prima di uscire, verificare con valgrind che questo sia avvenuto (se correttamente installato sul vostro pc).

Soluzione:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct El {
    int info;
    struct El *next;
} ElementoListaInt;
typedef ElementoListaInt* ListaDiInteri;

main(){
    ListaDiInteri a, b, c;

    a = malloc(sizeof(ElementoListaInt));
    b = malloc(sizeof(ElementoListaInt));
    c = malloc(sizeof(ElementoListaInt));

    /* una volta allocato lo spazio, i tre elementi non sono ancora una lista:
    vanno collegati tra loro. */

    a->info=1;
    a->next=b;

    b->info=2;
    b->next=c;

    /* come dicevamo, l'ultimo elemento (coda) *DEVE* avere NULL nel campo next! */
    c->info=3;
    c->next=NULL;

    /* anche se in questo caso non e' importante, bisogna stare attenti all'ordine
    con cui una lista viene distrutta: e' infatti illegale accedere ai campi di
    un'area di memoria che sia stata liberata con free.
    Il che significa che se non avessi da parte il puntatore di b e c, se eseguo
    la free(a) per prima, dopo non potrei raggiungere piu' b e c e quindi non
    potrei distruggerli! */

    free(c);
    free(b);
    free(a);
}
```

